orm2 中文文档

wizardforcel

Published
with GitBook

Table of Contents

Introduction	0
orm2 中文文档	1
连接到数据库	2
设置	3
定义模型	4
模型属性	4.1
模型验证器	4.2
模型钩子	4.3
模型缓存	4.4
定义关联	5
hasOne（多对一关系）	5.1
hasMany（多对多关系）	5.2
extendsTo（一对一关系）	5.3
同步和删除模型	6
查找记录	7
创建和更新记录	8
聚合	9

orm2 中文文档

译者：飞龙

来源：node-orm2 Wiki

协议：CC BY-NC-SA 3.0

orm2 中文文档

> 译者：飞龙
>
> 来源：Object Relational Mapping

安装

```
npm install orm
```

所支持的Node.js版本

支持 0.8, 0.10, 0.12, iojs-1.5 。

`0.10.x` ， `0.12.x` 和 `iojs-1.5` 版本的测试在 Travis CI 上运行。如果你想要的话，可以在本地运行测试：

```
npm test
```

DBMS 支持

- MySQL & MariaDB
- PostgreSQL
- Amazon Redshift
- SQLite
- MongoDB （beta版，到现在为止缺少聚合）

特性

- 创建模型，同步，删除，批量创建，获取，查找，移除，计数，聚合函数
- 创建模型的关联，查找，检查，创建和移除
- 定义自定义的验证器（有一些内建的验证器，会在保存之前检查实例的属性 -- 详见enforce
- 模型实例的缓存和一致性（两次获取表中的一行，获取到相同的对象，修改其中一个就是修改全部）
- 插件：MySQL FTS，Pagination （分页），Transaction （事务），Timestamps （时间戳），Migrations （迁移）

介绍

这是一个 Node.js 对象关系映射模块。

示例：

```
var orm = require("orm");

orm.connect("mysql://username:password@host/database", function (er
  if (err) throw err;

    var Person = db.define("person", {
        name      : String,
        surname   : String,
        age       : Number, // FLOAT
        male      : Boolean,
        continent : [ "Europe", "America", "Asia", "Africa", "Austr
        photo     : Buffer, // BLOB/BINARY
        data      : Object // JSON encoded
    }, {
        methods: {
            fullName: function () {
                return this.name + ' ' + this.surname;
            }
        },
        validations: {
            age: orm.enforce.ranges.number(18, undefined, "under-ag
        }
    });

    // add the table to the database
    db.sync(function(err) {
        if (err) throw err;

        // add a row to the person table
        Person.create({ id: 1, name: "John", surname: "Doe", age: 2
            if (err) throw err;

                // query the person table by surname
                Person.find({ surname: "Doe" }, function (err, peop
                    // SQL: "SELECT * FROM person WHERE surname =
                    if (err) throw err;

                    console.log("People found: %d", people.length);
                    console.log("First person: %s, age %d", people[

                    people[0].age = 16;
                    people[0].save(function (err) {
                        // err.msg = "under-age";
                });
            });

        });
    });
});
```

Promise

你可以使用开启Promise的包装库。

Express

如果你使用了Express，你可能想使用这一简单的中间件，使集成变得更容易。

```
var express = require('express');
var orm = require('orm');
var app = express();

app.use(orm.express("mysql://username:password@host/database", {
    define: function (db, models, next) {
        models.person = db.define("person", { ... });
        next();
    }
}));
app.listen(80);

app.get("/", function (req, res) {
    // req.models is a reference to models used above in define()
    req.models.person.find(...);
});
```

你可以多次调用 `orm.express` 来获取多个数据库的连接。在多个连接之间定义的模型会在 `req.models` 中连接。不要忘记在 **`app.use(app.router)`** 之前使用它，最好在你的公共素材文件夹之后。

示例

请见 `examples/anontxt` ，里面有一个基于express的应用示例。

连接到数据库

> 译者：飞龙
>
> 来源：Connecting to Database

在连接之前，你需要一个受支持的驱动。下面是一些测试过的驱动及其版本，把你所需要的加入到 `package.json` 中。

驱动	npm 包	版本
mysql	mysql	2.0.0-alpha9
postgres redshift	pg	2.6.2 [1]
sqlite	sqlite3	2.1.7
mongodb	mongodb	1.3.19

[1] 如果你要连接到**Heroku**，请使用版本**2.5.0**。

这些是测试过的版本，使用其它的版本（新的或者旧的）带来的风险由你自己承担。

例如，使用MySQL要这样做：

```
$ npm install --save mysql@2.0.0-alpha8
```

你可以传递一个URL字符串来连接数据库，其中scheme为受支持的驱动，或者你可以传递一个带有连接参数的 `Object` 。

```
var orm = require('orm');

orm.connect('mysql://root:password@localhost/test', function(err, d
  if (err) return console.error('Connection error: ' + err);

  // connected
  // ...
});
```

回调函数只在连接建立成功（或失败）时调用。如果你愿意的话，可以不传入回调函数，而是监听 `connect` 事件。

```
var orm = require('orm');

var db = orm.connect('mysql://root:password@localhost/test');

db.on('connect', function(err) {
  if (err) return console.error('Connection error: ' + err);

  // connected
  // ...
});
```

连接URL遵循下面的语法： `driver://username:password@hostname/database?option1=value1&opti`

可选参数为：

- `debug` （默认为 `false` ）：将连接输出到控制台；
- `pool` （默认为 `false` ）：使用驱动内建的组件管理连接池（仅对 `mysql` 和 `postgres` 有效）；
- `strdates` （默认为 `false` ）：以字符串形式保存日期（仅对 `sqlite` 有效）；
- `timezone` （默认为 `local` ）：在数据库中使用指定的时区储存日期（仅对 `mysql` 和 `postgres` 有效）；

`debug` 和 `pool` 也可以使用 `settings` 对象来设置。

连接到多个数据库

ORM模型受数据库连接约束，所以如果你需要“多租户”，即连接到不同的服务器或数据库，你可以使用像下面这样的方法：

```
// db.js
var connections = {};

function setup(db) {
  var User = db.define('user', ...);
  var Shirt = db.define('shirt', ...);
  Shirt.hasOne('user', User, ...);
}

module.exports = function(host, database, cb) {
  if (connections[host] && connections[host][database]) {
    return connections[host][database];
  }

  var opts = {
    host:     host,
    database: database,
    protocol: 'mysql',
    port:     '3306',
    query:    {pool: true}
  };

  orm.connect(opts, function(err, db) {
    if (err) return cb(err);

    connections[host] = connections[host] || {};
    connections[host][database] = db;
    setup(db);
    cb(null, db);
  });
};

// somewhere else, eg, middleware

var database = require('./db');

database('dbserver1', 'main', function(err, db) {
  if (err) throw err;

  db.models.user.find({foo: 'bar'}, function(err, rows) {
    // ...
  });
});
```

连接是被缓存的，所以模型在每个服务器+数据库上面只会定义一次。由于我们使用了连接池，我们并不需要担心用完所有的连接，而且我们可以一次性执行多个查询。

问题排除

如果你在连接MySQL数据库的时候遇到了如下错误：

```
Error: connect ECONNREFUSED
    at errnoException (net.js:670:11)
    at Object.afterConnect [as oncomplete] (net.js:661:19)
```

你可以尝试添加 `socketPath` 参数：

```
var db = orm.connect({
    host:     'localhost',
    database: 'database',
    user:     'user',
    password: 'pass',
    protocol: 'mysql',
    socketPath: '/var/run/mysqld/mysqld.sock',
    port:     '3306',
    query:    {pool: true, debug: true}
});
```

设置

> 译者：飞龙
>
> 来源：Settings

设置用于储存键值对。设置对象是 `orm` （默认值）上的实例，之后会为每个 `db` 连接和每个定义过的 `Model` 建立快照。所以 `orm.settings` 上的更改只会作用于更改之后建立的连接，而 `db.settings` 会作用于更改之后定义的模型。

```
var orm = require("orm");

orm.settings.set("some.deep.value", 123);

orm.connect("....", function (err, db) {
    // db.settings is a snapshot of the settings at the moment
    // of orm.connect(). changes to it don't affect orm.settings

    console.log(db.settings.get("some.deep.value")); // 123
    console.log(db.settings.get("some.deep"));       // { value: 12

    db.settings.set("other.value", { some: "object" });

    console.log(db.settings.get("other.value"));     // { some: "ob
    console.log(orm.settings.get("other.value"));    // undefined
});
```

默认设置的结构是这样的：

```
var Settings = {
    properties : {
        primary_key     : "id",
        association_key : "{name}_{field}",
        required        : false
    },
    instance   : {
        cache           : true,
        cacheSaveCheck  : true,
        autoSave        : false,
        autoFetch       : false,
        autoFetchLimit  : 1,
        cascadeRemove   : true,
        returnAllErrors : false
    },
    connection : {
        reconnect       : true,
        pool            : false,
        debug           : false
    }
};
```

设置	描述
`properties.primary_key`	在没有定义id属性的模型中，定义主键的名称
`properties.association_key`	关联键的属性名称（例如 `user_id` ）
`properties.required`	属性是否拥有默认行为
`instance.cache`	实例是否应该被缓存 （并不是真的缓存，和单例模式相关）
`instance.cacheSaveCheck`	被缓存的对象是否应该从缓存中返回 （不要修改这个设置，除非你知道自己在做什么）
`instance.autoSave`	如果开启的话，修改实例的任何属性时会自动保存
`instance.autoFetch`	是否需要自动获取关联
`instance.autoFetchLimit`	如果开启了自动获取关联，这个设置是获取关联的深度
`instance.cascadeRemove`	删除实例时是否要删除关联
`instance.returnAllErrors`	如果开启，实例保存时会记录下所有的错误并以数组形式返回，而不是遇到第一个错误就中止并返回
`connection.reconnect`	连接失效时是否尝试重新连接
`connection.pool`	是否使用驱动带有的连接池（如果支持的话）
`connection.debug`	向控制台打印带颜色的查询信息

定义模型

> 译者：飞龙
>
> 来源：Defining Models

在连接之后，你可以使用连接对象（ `db` ）来定义你的模型。你需要指定模型的名称，一个用于描述的属性和一些（可选的）选项。下面是一个简短的例子：

```
var Person = db.define('person', {
  id:      {type: 'serial', key: true}, // the auto-incrementing pr
  name:    {type: 'text'},
  surname: {type: 'text'},
  age:     {type: 'number'}
}, {
  methods : {
    fullName: function() {
      return this.name + ' ' + this.surname;
    }
  }
});
```

这个模型叫做 `person` （通常也是数据库里面表的名称），它有三个属性（ `name` 和 `surname` 为文本， `age` 为数值）。如果你自己不指定任何键的话，默认的 `id: { type: 'serial', key: true }` 会添加进来。在这个例子中，有个模型方法叫做 `fullName` 。下面是这个模型的使用方法的示例：

```
Person.get(73, function(err, person) {
  if (err) throw err;

  console.log('Hi, my name is ' + person.fullName());
});
```

这会获取 `id=73` 的 `person` 对象，并且打印出它的名字和姓氏。其它类型的可用属性请见这里。

API

```
/**
 * @param {Object} props Property definitions
 * @param {Object} opts Options
 */
db.define(props, opts)
```

`db.define()` 接收的第一个对象（第二个参数）被称为属性对象，它定义了所有的属性。

第二个对象指定了额外的选项：

选项名称	类型	描述
`collection`	`String`	覆写数据库中表的名称
`methods`	`Object`	模型实例上的额外方法，它会被设置到实例上。
`hooks`	`Object`	用户定义的钩子或回调
`validations`	`Object`	用户定义的验证器
`id`	`Array`	为了支持在 `properties` 上设置 `key: true` 而不提倡使用
`cache`	`Boolean`	允许你开启或者禁用单例行为。它叫做 `cache` ，但是和缓存毫无关系。
`autoSave`	`Boolean`	不推荐。在属性修改时自动保存模型。
`autoFetch`	`Boolean`	是否自动获取关联
`autoFetchLimit`	`Number`	自动获取关联的深度
`cascadeRemove`	`Boolean`	删除实例时是否要删除关联

模型属性

> 译者：飞龙
>
> 来源：Model Properties

模型和一些关联具有一个或多个属性，每个属性有类型以及一些可选设置，你可以自行选择它们（或使用默认设置）。

类型

受支持的类型是：

- `text` ：文本字符串；
- `number` ：浮点数。你可以指定 `size` 为 `2 | 4 | 8` ；
- `integer` ：整数。你可以指定 `size` 为 `2 | 4 | 8` ；
- `boolean` ： `true` 或 `false` 的值；
- `date` ：日期对象。你可以指定 `time` 为 `true` ；
- `enum` ：一个备选列表中的值；
- `object` ：JSON对象；
- `point` ：N维的点（不被广泛支持）；
- `binary` ：二进制数据；
- `serial` ：自增长的整数，用于主键。

每个类型都有额外的选项。这个模型定义使用了它们中的绝大多数：

```
var Person = db.define("person", {
    name    : { type: "text", size: 50 },
    surname : { type: "text", defaultValue: "Doe" },
    male    : { type: "boolean" },
    vat     : { type: "integer", unique: true },
    country : { type: "enum", values: [ "USA", "Canada", "Rest of t
    birth   : { type: "date", time: false }
});
```

所有类型都支持 `required` （布尔值）， `unique` （布尔值）和 `defaultValue` （文本）。文本类型也支持最大尺寸（数值）和 `big` （布尔值，用于非常长的字符串）。数值类型是浮点数，支持 `size` （数值，字节大小）和 `unsigned` （布尔值）。日期类型支持 `time` （布尔值）。

要注意**8**字节的数值有其局限性。

如果你打算用默认选项，你可以使用原生类型来指定属性类型：

```
var Person = db.define("person", {
    name    : String,
    male    : Boolean,
    vat     : Number, // FLOAT
    birth   : Date,
    country : [ "USA", "Canada", "Rest of the World" ],
    meta    : Object, // JSON
    photo   : Buffer  // binary
});
```

将ORM字段映射到不同名称的数据库列中

```
var Person = db.define("person", {
    name    : { type: 'text', mapsTo: 'fullname' }
});
```

ORM属性 `name` 映射 `person` 表的 `fullname` 列。

自定义类型

你可以向ORM添加你自己的类型，像这样：

```
db.defineType('numberArray', {
  datastoreType: function(prop) {
    return 'TEXT'
  },
  // This is optional
  valueToProperty: function(value, prop) {
    if (Array.isArray(value)) {
      return value;
    } else {
      return value.split(',').map(function (v) {
        return Number(v);
      });
    }
  },
  // This is also optional
  propertyToValue: function(value, prop) {
    return value.join(',')
  }
});
var LottoTicket = db.define('lotto_ticket', {
  numbers: { type: 'numberArray' }
});
```

一些可用的高级自定义类型，能够让你像 PostGIS 那样使用模型。请见这个 spec
。

模型验证器

> 译者：飞龙
>
> 来源：Model Validations

Enforce模块用于验证数据。对于使用以前的验证器的用户，还可以继续使用，它们中的一部分整合到了enforce，剩余部分还没有。推荐你开始使用 `orm.enforce` 来取代 `orm.validators` 。可用的验证器的列表请见node-enforce。

`unique` 验证器也构建于ORM中，可以这样来访问：

```
name: orm.enforce.unique("name already taken!")
name: orm.enforce.unique({ scope: ['age'] }, "Sorry, name already t
name: orm.enforce.unique({ ignoreCase: true }) // 'John' is same as
```

你可以为模型的每个属性定义验证器。对于每个属性，你可以定义一个或多个验证器。你也可以使用预定义的验证器，或者自己新建。

```
var Person = db.define("person", {
    name : String,
    age  : Number
}, {
    validations : {
        name : orm.enforce.ranges.length(1, undefined, "missing"),
        age  : [ orm.enforce.ranges.number(0, 10), orm.enforce.list
    }
});
```

上面的代码限定了 `name` 的长度必须在1和undefined之间（undfined表示任意值），以及 `age` 必须在0和10（闭区间）之间，而且是列出的值之一。这个例子或许没有意义，但是足够解释了。

保存一个对象的时候，如果由任何一个验证器验证失败，你都会得到一个带有属性名称和验证错误描述的 `error` 对象。这个描述可以帮助你弄清楚发生了什么。

```
var John = new Person({
    name : "",
    age : 20
});
John.save(function (err) {
    // err.field = "name" , err.value = "" , err.msg = "missing"
});
```

在第一个验证器验证失败之后，验证就停止了。如果你想要验证每个属性并且返回所有验证错误，你可以在全局或局部设置中更改这一行为：

```
var orm = require("orm");

orm.settings.set("instance.returnAllErrors", true); // global or..

orm.connect("....", function (err, db) {
    db.settings.set("instance.returnAllErrors", true); // .. local

    // ...

    var John = new Person({
        name : "",
        age : 15
    });
    John.save(function (err) {
        assert(Array.isArray(err));
        // err[0].property = "name" , err[0].value = "" , err[0].ms
        // err[1].property = "age"  , err[1].value = 15 , err[1].ms
        // err[2].property = "age"  , err[2].value = 15 , err[2].ms
    });
});
```

模型钩子

> 译者：飞龙
>
> 来源：Model Hooks

如果你想要监听发生在模型实例上的事件，你可以附带一个函数，它会在发生时调用。

现在支持下面这些事件：

- `afterLoad` ：（无参数）加载和准备所用实例之后；
- `afterAutoFetch` ：（无参数）自动获取关联（如果有的话）之后，无论有没有关联都会触发；
- `beforeSave` ：（无参数）尝试保存之前；
- `afterSave` ：（bool success）保存之后；
- `beforeCreate` ：（无参数）尝试保存新的实例之前（优先于 `beforeSave` ）；
- `afterCreate` ：（bool success）保存新的实例之后；
- `beforeRemove` ：（无参数）尝试删除实例之前；
- `afterRemove` ：（bool success）删除实例之后；
- `beforeValidation` ：（无参数）在所有验证之前，优先于 `beforeCreate` 和 `beforeSave` 。

所有钩子函数调用时， `this` 为对应的实例，所以你可以访问到与之相关的任何东西。

对于所有 `before*` 钩子，你可以添加一个额外的参数到钩子函数中。这个函数用来告诉钩子应该继续执行下去还是中断。你或许已经从Express的工作流中熟悉了这一点。下面是一个示例：

```
var Person = db.define("person", {
    name    : String,
    surname : String
}, {
    hooks: {
        beforeCreate: function (next) {
            if (this.surname == "Doe") {
                return next(new Error("No Does allowed"));
            }
            return next();
        }
    }
});
```

这个工作流允许你在调用 `next` 之前执行异步的操作。如果你不打算使用 `next` 就不要把它定义为参数，否则会阻塞工作流。

常见问题

一个常见问题涉及到在钩子内部的嵌套回调中访问 `this` 。这个问题的原因是， `this` 对象仅仅在顶级钩子函数的作用域内是有效的，而在回调中会有各种不同的值。要解决这一问题，可以创建一个对象保存 `this` 的引用，并且在回调中用它来访问模型的属性。

示例

```
var Person = db.define("person", {
    name    : String,
    surname : String
}, {
    hooks: {
        beforeCreate: function (next) {
            var _this = this;
            checkName(this, function(err, result)) {
                if(err) return next(err);
                _this.name = result.name;
                _this.surname = result.surname;
                next();
            }
        }
    }
});
```

模型缓存

定义关联

> 译者：飞龙
>
> 来源：Defining Associations

关联是一个或多个模型之间的关系。

关联的类型：

- hasOne（多对一）
- hasMany（多对多）
- extendsTo（一对一）

hasOne（多对一关系）

> 译者：飞龙
>
> 来源：hasOne

hasOne关联是一种多对一的关系，意思是你定义的模型可以有多个实例指向一个其它的实例（所属相同模型或不同模型）。

用法

```
Animal.hasOne(association_name [, association_model [, options ] ])
```

描述

- `association_name` 是两个模型之间的关系名称‘
- `association_model` 是要关联的另一个模型（如果没有定义，假设为同一个模型，大多数情况下这可能不是你想要的）；
- `options` 是一个对象，拥有一些和关联有关的，你可以调整的属性，比如自动获取，再比如表（SQL中）或者集合（MongoDB中）的名称。

示例

```
Animal.hasOne("owner", Person);
```

在背后，这条语句意思是 `Animal` 集合拥有一个属性 `owner_id` （这个名称可以通过选项来修改， `{field: 'ownerid'}` ），它会指向 `Person` 集合的某个人。如果关联并不是必须的，则可以为空。

这个关联也会创建一些额外的便利方法（叫做关联访问器）来帮助你管理它。访问器的名称也可以修改（同上，在选项里面），默认情况下，它们会拥有和关联名称相似的名称。例如，下面的代码展示了可以做类似这样的事情：

```
// assuming John is a Person..
Animal.find({ name: "Deco" }).first(function (err, Deco) {
    Deco.setOwner(John, function (err) {
        // John is now the owner of Deco
    });
});
```

其它的访问器：

- `getOwner(callback)` - 获取关联的所有者
- `hasOwner(callback)` - （在回调中）返回这个动物是否拥有所有者
- `removeOwner(callback)` - 移除和所有者的关联关系（如果存在的话）

关联反转

有时你希望通过对面的模型来访问关联。在上面的例子中，是通过 `Person` 。你可以向关联传递一个选项来实现它。

```
Animal.hasOne('owner', Person, { reverse: "pets" });
```

之后，每个 `person` 实例都有有两个便利方法：

- `getPets(callback)` - 获取所有和这个人有关联的动物
- `setPets(cat, dog, callback)` - 移除所有和这个人有关联的动物，并且添加猫和狗

hasMany（多对多关系）

> 译者：飞龙
>
> 来源：hasMany

hasMany

是多对多的关系（包括连接表）。

例如：`Patient.hasMany('doctors', Doctor, { why: String }, { reverse: '`。

病人可以拥有许多不同的医生。每个医生可以拥有许多不同的病人。

当你调用 `Patient.sync()` 时，会创建一个连接表 `patient_doctors` 。

列名称	类型
patient_id	Integer
doctor_id	Integer
why	varchar(255)

下列函数是可用的：

```
// 获取所有关联医生的列表
patient.getDoctors(function(err, doctors) {
  // ...
});

// 向连接表中增加记录
patient.addDoctors([phil, bob], function(err) {
  // ...
});

// 移除连接表中的现有记录，并增加新的
patient.setDoctors([phil, nephewOfBob], function(err) {
  // ...
});

// 检查是否某个病人关联了指定的医生
patient.hasDoctors([bob], function(err, patientHasBobAsADoctor) {
  // because that is a totally legit and descriptive variable name
  if (patientHasBobAsADoctor) {
    // ...
  } else {
    // ...
  }
});

// 从连接表中移除指定记录
patient.removeDoctors([bob], function(err) {
  // ...
});

// 并且所有医生都有自己的方法来获取病人
bob.getPatients(function(err, patients) {
  if (patients.indexOf(you) !== -1) {
    // woot!
  } else {
    // ...
  }
});

// 以及其他
```

要把医生关联到病人：

```
patient.addDoctor(surgeon, {why: 'remove appendix'}, function(err)
  // ...
});

// or...
surgeon.addPatient(patient, {why: 'remove appendix'}, function(err)
  // ...
});
```

这样会添加 `{patient_id: 4, doctor_id: 6, why: "remove appendix"}` 到连接表中。

API

```
Model.hasMany(
  name,        // String. 关联名称
  otherModel, // Model. 要关联的模型
  extraProps, // Object. 在连接表上出现的额外属性
  opts        // Object. 关联的选项
);
```

选项

选项名称	类型	描述
autoFetch	Boolean	默认为 `false` 。如果为 `true` ，关联将会自动被获取。
autoFetchLimit	Number	默认为 `1` 。自动获取的深度。
key	Boolean	默认为 `false` （由于历史原因）。如果为 `true` ，表中外键的列会形成一个组合键。
mergeTable	String	连接表的自定义名称
mergeId	String	代表当前模型那一列的自定义名称
mergeAssocId	String	代表另一个模型那一列的自定义名称
reverse	String	默认为 `false` 。如果为 `true` ，关联可以通过另一个模型使用指定方法获取到。
getAccessor	String	默认为 `'get' + Name` 。允许重命名关联访问器。
setAccessor	String	默认为 `'set' + Name` 。允许重命名关联访问器。
hasAccessor	String	默认为 `'has' + Name` 。允许重命名关联访问器。
delAccessor	String	默认为 `'del' + Name` 。允许重命名关联访问器。
addAccessor	String	默认为 `'add' + Name` 。允许重命名关联访问器。

extendsTo（一对一关系）

> 译者：飞龙
>
> 来源：extendsTo

你可能想把可选的属性分割到另一个表中。每个扩展都会是一个新的表，其中每一行的唯一标识符是主模型实例的id。

例如：

```
var Person = db.define("person", {
    name : String
});
var PersonAddress = Person.extendsTo("address", {
    street : String,
    number : Number
});
```

这样会创建 `person` 表，带有 `id` 和 `name` 列。扩展行为会创建 `person_address` 表，带有 `person_id` ， `street` 和 `number` 列。 `Person` 模型中可用的方法类似于 `hasOne` 关联。这个例子中，你可以调用 `.getAddress(cb)` ， `.setAddress(Address, cb)` 以及其他。

注意：你并不需要保存 `Person.extendsTo` 的返回值，它返回了一个扩展模型。你可以使用它来直接查询扩展表（甚至查找相关的模型），但是这完全取决于你。如果你只希望通过原模型来访问它的话，可以丢弃返回值。

同步和删除模型

> 译者：飞龙
>
> 来源：Syncing and dropping models

同步是一项功能方法，可以在数据库里为你的模型和关联创建所需的表来工作。现存的表并不会被替换，它们只会在不存在的时候被创建。

同步有两种方式：

1. 调用 `Model.sync(cb)` 会仅仅同步指定模型
2. 调用 `db.sync(cb)` 会同步所有模型

删除是一个类似的方法，但是它会删掉你模型涉及的所有表，即使不是ORM创建的。删除也有两种方式。

```
var orm = require("orm");

orm.connect("....", function (err, db) {
    var Person = db.define("person", {
        name : String
    });
    var Pet = db.define("pet", {
        name : String
    });

    db.drop(function () {
        // 从指定模型中删除所有表（Person和Pet）

        Person.sync(function () {
            // 为Person模型创建表
        });
    });
});
```

查找记录

> 译者：飞龙
>
> 来源：Finding items

`find`

查找匹配标准的记录，可以链式查询（见下文）：

```
Person.find({status:'active'}, function(err, results) {
  // ...
});
```

你也可以限制结果的个数，这条语句限制结果为10个：

```
Person.find({status:'active'}, 10, function(err, results) {
  // ...
});
```

`Person.all` 是 `Person.find` 的别名。

`get`

通过主键来查找记录。

```
Person.get(1, function(err, person) {
  // ...
});
```

`one`

只查找一个记录，和 `find` 的语法相似。

```
Person.one({status:'active'}, function(err, person) {
  // ...
});
```

`count`

获取所匹配记录的数量。

```
Person.count({status:'active'}, function(err, activePeopleCount) {
  // ...
});
```

`exists`

测试匹配你的条件的记录是否存在。

```
Person.exists({id:1, status:'active'}, function(err, personIsActive
  // ...
});
```

过滤和排序

我们接受两个对象来执行过滤（第一个）和聚合（第二个）。聚合对象接受 `limit` ， `order` 和 `groupBy` 。

https://github.com/dresende/node-orm2/blob/v2.1.20/lib/AggregateFunctions.js#L36

```
Person.find({status:'active'}, {limit:10}, function(err, res) {

});
```

`find` / `count` / `one` 等方法的条件查询

所有以逗号分隔的键值对在查询中都会以 `AND` 连接。你可以把逻辑运算符放在一系列条件的前面。

```
Person.find({or:[{col1: 1}, {col2: 2}]}, function(err, res) {
  // res 为 col1 == 1 或者 col2 == 2 的 Person
});
```

使用 `IN` 来查找

`sql-query` （取决于SQL引擎）会自动将数组视为基于 `IN` 的查询。

https://github.com/dresende/node-sql-query/blob/v0.1.23/lib/Where.js#L172

```
Person.find({id: [1, 2]}, function(err, persons) {
  // 查找 id 是 1 或者 2 的 Person （例如 WHERE id IN (1, 2) ）
});
```

创建和更新记录

> 译者：飞龙
>
> 来源：Creating and Updating Items

创建

```
var newRecord = {};
newRecord.id = 1;
newRecord.name = "John"
Person.create(newRecord, function(err, results) {
 ...
});
```

保存

```
Person.find({ surname: "Doe" }, function (err, people) {
    // SQL: "SELECT * FROM person WHERE surname = 'Doe'"

    console.log("People found: %d", people.length);
    console.log("First person: %s, age %d", people[0].fullName(), p

    people[0].age = 16;
    people[0].save(function (err) {
        // err.msg = "under-age";
    });
});
```

聚合

> 译者：飞龙
>
> 来源：Aggregation

如果你需要从一个模型中获取一些聚合值，你可以使用 `Model.aggregate()` 。下面通过一个例子来展示：

```
Person.aggregate({ surname: "Doe" }).min("age").max("age").get(func
    console.log("The youngest Doe guy has %d years, while the oldes
});
```

可以传递一个含有属性的 `Array` 来选择仅仅保留一小部分属性。方法也会接收一个 `Object` 来定义条件。

下面是一个展示如何使用 `.groupBy()` 的例子：

```
// 和 "select avg(weight), age from person where country='someCount
Person.aggregate(["age"], { country: "someCountry" }).avg("weight")
    // stats 是一个数组，每个记录都有 'age' 和 'avg_weight' 属性
});
```

基本的 `.aggregate()` 方法

- `limit()` ：你可以传递一个数值作为个数，或者两个数值分别作为偏移和个数
- `order()` ：和 `Model.find().order()` 相同

额外的 `.aggregate()` 方法

- `min`
- `max`
- `avg`
- `sum`
- `count` （它有一个快捷方式 - `Model.count` ）

有更多的聚合函数是依赖于驱动的（比如数学函数）。